User-Defined Datatypes

Mattox Beckman

beckman@iit.edu

Illinois Institute of Technology

Even more powerful than the idea of recursion is the idea of abstraction—which is a lot easier to implement if you can make your own types. Your goal after this lecture is to...

- Be able to describe the record type and the disjoint type.
- Be able to give an example of how each one is used.
- Be able to draw a memory diagram that is the result of allocating a variable of a disjoint type.
- Be able to use the `match/with` syntax to deconstruct a disjoint type.
- Be able to describe how data-structures in a functional language “recycle” data from previous versions of the structure when a modification has been made.

- Complex numbers have the form $a + bi$, where $i \equiv \sqrt{-1}$
- Addition: $(a + bi) + (c + di) = (a + c) + (b + d)i$
- Multiplication: $(a + bi) \times (c + di) = ac - bd + (ad + bc)i$

```
# let cadd (a,b) (c,d) = a +. c, b +. d
# let cmul (a,b) (c,d) = a *. c -. b *. d,
                         a *. d +. b *. c
# cmul (3.0,4.0) (1.0,4.4);;
- : float * float = -14.6, 17.2
```

We could use tuples to represent complex numbers, like this. (What are the types of these functions?) Why might this be a bad idea?

`type` *name* = { *name* : *type* [; *name* : *type* ...] }

Name Labels

```
# type complex = { re : float; im : float };;
type complex = { re : float; im : float; }
# let cadd x y = { re = x.re +. y.re;
                   im = x.im +. y.im };;
val cadd : complex -> complex -> complex = <fun>
```

- Fields are accessed with dot notation.
- Note that *field names must be unique*.

Activity

1. Write a function to multiply complex numbers. Use the type definition we had before.
2. What would the type defintion for a phone book entry look like? Assume we want to store a name, address, and phone number, all as strings.

Answer

1. Write a function to multiply complex numbers. Use the type definition we had before.

```
let mult a b = { re = a.re * b.re - a.im * b.im;
                 im = a.re * b.im + b.re * a.im; }
```

2. What would the type defintion for a phone book entry look like? Assume we want to store a name, address, and phone number, all as strings.

```
type address = {name : string; address : string;
                phone : string };;
```


Simple Type Definitions

`type` *name* = *Name* [`of` *type*] [| *Name* [`of` *type*] . . .]

Name Constructors Arguments

- Note: Constructor names must be capitalized.
- Constructor names also must be unique.

```
# type contest = Rock | Scissors | Paper
# type velocity = MeterPerSecond of float
                | FeetPerSecond of float;;
```


Example of `contest` and `velocity`

```
# let winner a b =
    match a,b with
    | Rock,Scissors | Scissors,Paper | Paper,Rock
        -> "Player 1"
    | Rock,Paper | Scissors,Rock | Paper,Scissors
        -> "Player 2"
    | _ -> "Tie";;
val winner : contest -> contest -> string = <fun>
# let thrust vel =  If only NASA had used OCaml!
    match vel with
    | FeetPerSecond x   -> x /. 3.28
    | MetersPerSecond x -> x;;
val thrust : velocity -> float = <fun>
```

Bug Alert

foo.ml

```
# type foo = Foo of int;;
# let showfoo (Foo n) = n;;
```

```
# #use "foo.ml";; This loads the file foo.ml
# let t1 = Foo 5;;
# showfoo t1;;
- : int = 5
# #use "foo.ml";; At this point you reload....
# showfoo t1;;
Characters 8-10:
This expression has type foo but is here used
with type foo
```

Why did this happen?

The Most Fun Datatypes are Recursive

```
# type mylist = Cons of int * mylist
                | Nil
# let rec mklist lst =
     match lst with
     | [] -> Nil
     | x::xs -> Cons (x, mklist xs);;
# let l1 = mklist [2;3;4];;
val l1 : mylist = Cons (2, Cons (3, Cons (4, Nil)))
```

- A recursive type without a recursive case is not really recursive.
- A recursive type without a base base is useless. Unless you're using Haskell, which supports infinite data-structures. But we're not going to talk about that here.

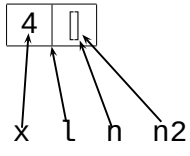
Type Constructors and Memory

- When a type constructor is invoked, it causes memory to be allocated.
  - Writing an integer
  - Writing `[]` or `::`
  - Using :: or `Cons`
- Writing down a variable does not cause memory to be allocated.

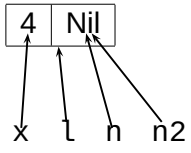
```
# let x = 4;;  allocates 4
# let n = [];;  allocates empty list
# let n2 = n;;  does NOT allocate memory
# let l = x::n;; A cons cell is allocated,
  but not the 4 or the empty list
```


Memory Diagram

```
# let x = 4;;  allocates 4
# let n = [];;  allocates empty list
# let n2 = n;;  does NOT allocate memory
# let l = x::n;; A cons cell is allocated,
  but not the 4 or the empty list
```



```
# let x = 4;;  allocates 4
# let n = Nil;;  allocates Nil
# let n2 = n;;  does NOT allocate memory
# let l = Cons (x,n);; A cons cell is allocated,
  but not the 4 or a Nil
```



How would you write `length` and `inclist` for this type?


```
# let rec length lst =
     match lst with
     | Nil -> 0
     | Cons(x,xs) -> 1 + length xs;;
# length l1
- : int = 3
# let rec inclist lst =
     match lst with
     | Nil -> Nil
     | Cons(x,xs) -> Cons(x+1,inclist xs);;
# inclist l1;;
- : mylist = Cons (3, Cons (4, Cons (5, Nil)))
```



```
# let rec reverse lst =
    let rec aux lst acc =
      match lst with
      | Nil -> acc
      | Cons(x,xs) -> aux xs (Cons(x,acc)) in
    aux lst Nil;;
# reverse l1;;
- : mylist = Cons (4, Cons (3, Cons (2, Nil)))
```


OCaml supports parametric polymorphism, like templates in C++ or generics in Java.

```
# type 'a mylist = Cons of 'a * 'a mylist
                 | Nil
# let rec length xx =
     match xx with
     | Nil -> 0
     | Cons (_,xs) -> 1 + length xs;;
val length : 'a mylist -> int = <fun>
# length (Cons(3,Cons(4,Nil)));;
- : int = 2
# length (Cons("Hi",Cons("There",Nil)));;
- : int = 2
```

Activity

1. Write the type for a binary search tree. In this version, let a "node" be something that has data, and a "leaf" does not have any data. Use parametric polymorphism.
2. Write `find` and `add`.

Let's take these one at a time.

BST Type

Write the type for a binary search tree.

```
type 'a bst = Node of 'a * 'a BST * 'a BST
            | Leaf
```

Now write `find`.

Find

```
let rec find a xx =
  match xx with
  | Node (x,l,r) when x=a -> true
  | Node (x,l,r) -> if x<a then find a l
                           else find a r
  | Leaf -> false
```

Now write `add`. If the element is already added, return the value unchanged.

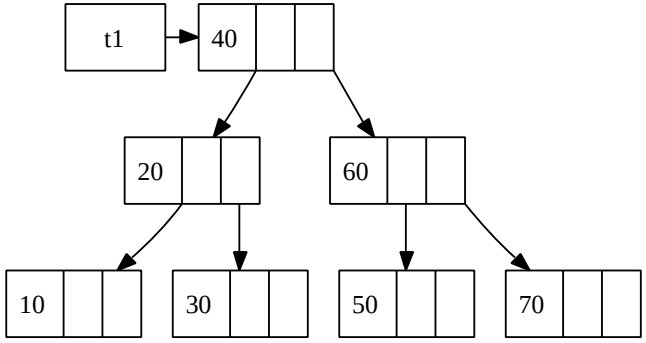
Add

```
let rec add a xx =
  match xx with
  | Node (x,l,r) when x=a -> xx
  | Node (x,l,r) -> if x<a then Node(x, add a l, r)
                           else Node(x, l, add a r)
  | Leaf -> Node(a,Leaf,Leaf)
```

What will be the result of

```
let t1 = add 10 (add 30 (add 50 (add 70
  (add 20 (add 60 (Node 40 Leaf Leaf))))));;
```

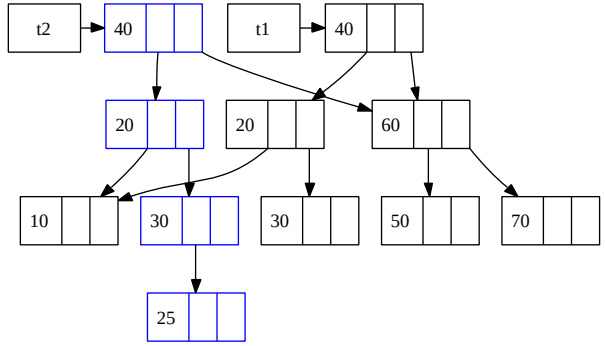
Before:



Now execute:

```
1 let t2 = add 25 t1;;
```

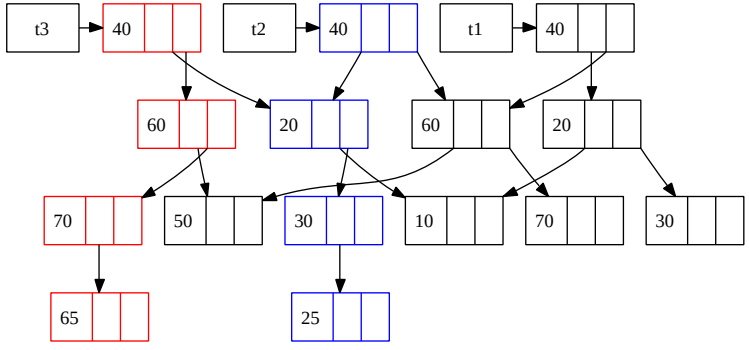

Before:



Now execute:

```
1 let t3 = add 65 t2;;
```


Before:



- This technique is called *functional-style updating*.
- Advantage: multiple versions of the data-structure can exist. This allows multiple versions, and consistent multiprocessing. Used to implement multi-level undo.
- Disadvantage: it takes more memory.

A very common type definition

```
# type 'a option = Some of 'a | None;;
# let rec getItem key lst =
    match lst with
    | [] -> None    note the type variables!
    | (k,v)::xs -> if key = k then Some v
                   else getItem key xs;;
# getItem 3 [2,"french hens"; 3,"turtle doves"];;
- : string option = Some "turtle doves"
# getItem 5 [2,"french hens"; 3,"turtle doves"];;
- : string option = None
```

What options would you have for this function if you did *not* have an `option` type?